REMARKS

Amendments to the Specification

The requested amendments to the specification have three purposes: 1) to refer to certain then pending applications with the numbers of their corresponding issued patents so that these references will be more readily accessible by readers; 2) to reduce redundancy; and 3) to correct a typographical error (the reference number 136 for the binary translator). These changes do not introduce new matter.

Claim Rejections Under 35 U.S.C. 103(a)

The Examiner rejected all of the original claims, that is, claims 1-24, under 35 U.S.C. 103(a) as being unpatentable over Yates, et al. (6,549,959 – "*Yates*") in view of statements on pages 1-3 of the applicant's specification.

In essence, concerning independent claim 1, the Examiner wrote that *Yates* discloses all of the main features of the applicant's invention except conversion of source instructions into target instructions (or instruction sequences) using binary translation, but that it would be obvious to combine *Yates*' teachings with the known mechanism of binary translation.

The applicant observes that, although *Yates*' system involves cross-architectural translation (between non-native x86 code and native "Tapestry" code), *Yates does* discuss binary translation. See, for example, *Yates*' section I. D. and col. 1, lines 60-65:

> In a third alternative, binary translation, a translator program takes the non-native binary (either a whole program or a program fragment) as input, and processes it to produce as output a corresponding binary in the native instruction set (a "native binary") that runs directly on the computer.

Even so, *Yates* fails to disclose a key novel and non-obvious feature of the invention as claimed. Although this feature was present in the original independent claims, the independent claims 1, 15 and 16 have now been amended to more clearly recite it, namely, "delaying handling of each asynchronous sensed exception until, and no later than, completion of execution of the target instruction sequence corresponding to the current source instruction when the asynchronous exception is

sensed." Claim 1 therefore now includes all the limitations of the originals claims 1-4 and 7.

The applicant's invention provides a mechanism that allows execution of a translated sequence to roll ***forward*** when an asynchronous exception arises during execution of the translated sequence. The amount of roll-forward is well-defined to be no more than to completion of execution of the translation of the current source instruction.

The importance of this guarantee of precise exception handling is discussed in the specification on page 25, lines 9-23:

> In this example, as well as other cases, the subsystems rely on a very precise guarantee, namely, that the monitor action will be processed at the latest at the next virtual machine instruction boundary. Certain subsystems, such as the graphics subsystem, rely on this guarantee for synchronization purposes. Others do not need to rely on the guarantee for correctness, but nonetheless benefit from it to improve performance. For example, it allows various device emulation subsystems such as the disk subsystem to raise the interrupt line of the virtual machine with minimal latency because only the current source instruction needs to complete execution.
>
> This differs from approaches used in the prior art, which typically do not offer such a guarantee. For example, Shasta, a distributed memory system built around a binary translator, also has a requirement to handle frequent asynchronous actions (in that case, cache-coherency invalidation). Unlike this invention, however, the Shasta system provides a less-precise guarantee that simply states that the asynchronous event will be handled, on average, within a small delay. This difference in the guarantees offered by the system has a profound implication on the implementation.

In *Yates*, code may be executed in three different contexts: 1) native Tapestry code, which can be executed as is; 2) non-native x86 code that is translated to native code in the TAXi binary translator 124; or 3) non-native code that must be referred to the hardware "converter" 136. It is only in the last two contexts that exceptions must be handled during execution of translated (or "converted") code. In every case that *Yates* describes (as opposed to merely mentions – see below), *Yates* discloses a roll-***back*** mechanism (emphasis added in each quotation):

Col.6, lines 32-34

Delivery of an interrupt may be deferred by a time sufficient to allow the thread to reach a checkpoint, or execution of the thread may be ***rolled back*** to a checkpoint...

Col. 18, lines 56-67

In general, in a fifty-third aspect, the invention features a method and computer for performance of the method. A first interpreter executes a program coded in an instruction set, the first interpreter being less than fully correct. A second, fully-correct interpreter, primarily in hardware, executes instructions of the instruction set. A monitor detects any deviation from fully-correct interpretation by the first interpreter, before any side-effect of the incorrect interpretation is irreversibly committed. When the monitor detects the deviation, execution is ***rolled back*** by at least a full instruction to a safe point in the program, and execution is re-initiated in the second interpreter.

Translated native Tapestry code runs faster than converter 136, and is used when translation can be guaranteed to be correct, or when any divergence can be caught and corrected. The execution of the TAXi code is monitored to detect violations of the optimistic assumptions, so that any deviation from correct emulation of the X86 can be detected. Either a pre-check can detect that execution is about to enter a region of translated code that can not be trusted to execute correctly, or hardware delivers an exception after the fact when the optimistic assumptions are violated. In either case, when correctness cannot be guaranteed, or for code that translator 124 does not know how to translate, execution of the translated native Tapestry code is aborted or ***rolled back*** to a safe check point, and execution is resumed in the hardware converter 136.

Col. 19, lines 14-26

Embodiments of the invention may include one or more of the following features. The first interpreter may include a software emulator, and/or a software binary translator. The second interpreter may interpret instructions in an instruction set not native to the computer. The software binary translator may operate concurrently with execution of the program to translate a segment less than the whole of the program. Continuing execution may include ***rolling back*** execution of the first interpreter by at least two full instructions. Continuing execution may include ***rolling back*** execution of the first interpreter from a state in which a number of distinct suboperations of several instructions have been intermixed by the first interpreter.

Col. 28, lines 25-39

Translated native Tapestry code runs faster than converter 136, and is used when translation can be guaranteed to be correct, or when any divergence can be caught and corrected. The execution of the TAXi code is monitored to detect violations of the optimistic assumptions, so that any deviation from correct emulation of the X86 can be detected. Either a pre-check can detect that execution is about to enter a region of translated code that can not be trusted to execute correctly, or hardware delivers an exception after the fact when the optimistic assumptions are

violated. In either case, when correctness cannot be guaranteed, or for code that translator 124 does not know how to translate, execution of the translated native Tapestry code is aborted or ***rolled back*** to a safe check point, and execution is resumed in the hardware converter 136.

Col. 41, lines 9-14

If an interrupt is directed to something within virtual X86 310, while TAXi code (a translated native version of a "hot spot" within an X86 program, see section I.D, supra, as indicated by the TAXi_Active bit 198 of the EPC) was running, then the interrupt is handled by case 353. Execution is ***rolled back*** to an X86 instruction boundary.

Col. 96, lines 27-34

If a memory reference goes through a segment descriptor whose TAXi Optimized Load bit 810 is One, and the reference resolves to non-well-behaved memory (D-TLB.ASI, address space ID, not equal to Zero), and PSW.TAXi_Active 198 is true, then a TAXi I/O exception is raised. The handler for the TAXi I/O exception ***rolls*** the execution context ***back*** to the last safety net checkpoint ...

Col. 99, lines 1-12

Consider an example, where the TAXi translator uses optimistic assumptions and CSE's two loads together then execution is ***rolled back*** to an instruction boundary, where all extended Tapestry state is dead. The TAXi code is abandoned, and the original X86 code is executed in converter 136.

Col. 100, lines 53-64

In some instances, this annotation mechanism may ***roll back*** execution by a considerable number of instructions, in order to establish a "safe" state, where all X86 instructions can either be assumed to have not started, or completed completely. The ***rollback*** mechanism avoids resuming execution from a state where a single side-effect may be applied twice.

The code may "checkpoint" itself, capturing a self-consistent state snapshot somewhat in the manner of a database system. Then, in the event of a fault in the TAXi code, execution can be ***rolled back*** to the checkpoint, and resumed in converter 136.

Col. 101, lines 3-5

In another alternative embodiment, case 354 can ***roll back*** X86 emulator 316 to the previous X86 instruction boundary.

Thus, in all these cases, *Yates* rolls execution back by one or more instructions to a previous source (x86) instruction boundary, or to a "checkpoint." Such roll-back means that when an asynchronous exception occurs, there may be no forward progress in the execution until after the exception is handled. This also means that subsystems

running any *Yates*-like mechanism will ***not*** have the "precise guarantee, namely, that the monitor action will be processed at the latest at the next virtual machine instruction boundary" (specification, page 25, lines 9-12, quoted more fully above). As mentioned in the same paragraph of the specification, this guarantee may be necessary for some subsystems for the purpose of synchronization, but is also advantageous in general.

The applicant observes that *Yates* does mention, as an aside and without further discussion, rolling forward execution in two places:

> Col. 19, lines 24-27
> Continuing execution may include ... allowing execution to progress forward to a checkpoint in the first interpreter.

> Col. 100, line 66, to col. 101, line 16
> Referring again to FIG. 3j, in one alternative embodiment, if this is an asynchronous interrupt, case 351 or 354 can allow X86 emulator 316 or converter 136, respectively, to progress forward to the next X86 instruction boundary, before delivering the interrupt In other alternative embodiments, in the case of asynchronous interrupts in cases 351, 353, and 354, the code can be allowed to progress forward to the next safety net checkpoint before delivering the exception.

When *Yates* here refers to progressing forward to "the next X86 instruction boundary" (col. 100, line 67, to col. 101, line 2), *Yates* refers to cases 351 and 354, which involve execution in the hardware converter 136 and x86 emulator 316, but ***not*** in the TAXi binary translator 124, which is "case 353" in Fig. 3J. Moreover, in all cases in which *Yates* mentions forward progress in the TAXi translator, he refers to progress forward to a ***checkpoint***, which is not guaranteed to be the end of the translated instruction sequence corresponding to the source instruction current at the time of sensing an asynchronous exception:

> Col. 6, lines 34-38:
> checkpoints being points in the execution of the thread where the amount of extended context, being the resources of the thread beyond those whose resource association with the thread is maintained by the thread scheduler, is reduced.

> Col. 100, lines 61-62:
> The code may "checkpoint" itself, capturing a self-consistent state snapshot somewhat in the manner of a database system.

Yates therefore does not guarantee precise exception handling, but rather, in contrast, merely "eventual" exception handling. This contrasts with the applicant's invention as claimed, which guarantees "delaying handling of each asynchronous sensed exception until, ***and no later than***, completion of execution of the target instruction sequence corresponding to the current source instruction when the asynchronous exception is sensed." Thus, as the applicant states on page 25, lines 27-28 of the specification: "Unlike approaches used in the prior art, this invention provides such a guarantee of precise handling even of asynchronous events."

Claim Rejections in View of Hohensee

The Examiner did not state under what section of 35 U.S.C. he rejected claims 7 and 9 in view of U.S. Patent 5,778,211 (Hohensee, et al.). It appears from the Examiner's wording, however, that he posits some combination of the teachings of *Yates* and *Hohensee*. Whereas *Yates* – unlike the applicant's invention as claimed – fails to guarantee precise handling of asynchronous exceptions, *Hohensee* fails to explain how to handle asynchronous exceptions at all.

Hohensee involves conversion of original floating point instructions of an emulated processor (such as processors with an x86 architecture), which has a delayed exception-handling model, into corresponding translated instructions that are executed on a host processor 11 (such as processors with the SPARC Version 9 architecture), which implements a precise exception-handling model. See, for example, col. 8, lines 35-39. The problem *Hohensee* addresses is how to handle floating point exceptions that arise during execution of a translated instruction or instruction sequence so that the precise-handling host system will faithfully emulate the behavior of the non-precise original architecture.

Hohensee discloses two ways to do this: In the first embodiment, a floating point exception flag 34 is set whenever a floating point exception occurs. Instructions that are inserted into every translated sequence then test the flag to determine whether the *previous* instruction sequence experienced a floating point exception. In a second embodiment, if a floating point exception occurs, then a memory page is marked as

being write protected. A store (write) instruction is then inserted into every translated instruction sequence. If a floating point exception occurs, then the *following* instruction sequence, upon attempting a write to the now-protected page, will cause a write protection fault to be generated, which is taken as indication that a floating point exception occurred in the *previous* sequence.

Regardless of which of *Hohensee's* embodiments one considers, the only type of exception-handling that *Hohensee* discusses handling is floating point exceptions. However, on col. 16, lines 17-24, *Hohensee* does mention in passing other types of exceptions that his mechanism could be used to handle:

> "... it will be appreciated that the invention may further be used in connection with handling of exceptions which are generated during a myriad of other types of instructions, including, for example, integer instructions, logical instructions and program flow control instruction."

All of the exceptions *Hohensee's* mechanism is designed to handle are therefore ***synchronous*** exceptions.

The applicant's independent claims all recite "delaying handling of each asynchronous sensed exception until, and no later than, completion of execution of the target instruction sequence corresponding to the current source instruction when the asynchronous exception is sensed." This is not applicable in the case of synchronous exceptions because it would not make sense to complete of execution of the target instruction sequence corresponding to the current source is a synchronous exception arises during such execution: If an overflow has occurred, or an attempted memory access using an invalid or impermissible address, then continued execution of the translated instructions could lead to undefined and potentially damaging consequences.

How *Yates* or *Hohensee* handle synchronous exceptions is unrelated to the novel way in which the applicant's invention precisely handles asynchronous exceptions. Even so, the applicant notes that it would not be obvious to combine the teachings of *Yates* and *Hohensee,* even when it comes to handling synchronous exceptions. Upon the occurrence of a synchronous exception, *Yates* rolls execution back to a safe point and he restores a previous state:

Yates col 41, lines 25-28 (emphasis added):

Else, the ***rollback*** was induced by a synchronous event, so TAXi will resume execution in converter 136, and the exception will be re-triggered, with EPC.ISA 194 indicating X86, and the exception will be handled by case 351.

Yates col. 100, lines 46-59 (emphasis added):

Thus, when a synchronous exception is to be exposed to the virtual X86, the TAXi run time system establishes a system state equivalent to that which would prevail at an X86 instruction boundary. Once state is ***restored*** to a precise instruction boundary, execution can be tendered to converter 136, which in turn can resume execution from that instruction boundary.

In some instances, this annotation mechanism may ***roll back*** execution by a considerable number of instructions, in order to establish a "safe" state, where all X86 instructions can either be assumed to have not started, or completed completely. The ***rollback*** mechanism avoids resuming execution from a state where a single side-effect may be applied twice.

In contrast, *Hohensee's* mechanism does not even detect that a synchronous exception has occurred until some point in the *following* translated instruction sequence. Rolling back execution, as in *Yates*, would mean that *Hohensee's* system might not even reach the point at which it would detect that a synchronous exception had occurred. Combining *Yates* and *Hohensee* would therefore not be obvious, since it would defeat the mechanisms in *Hohensee* for detection of synchronous exceptions.

Voluntary claim amendments

Claims 16-23 have been amended to remove certain "means for" formulations so as to more closely match the phraseology used in the specification and to avoid unwarranted limitation of the claims in future interpretations. Claim 23 has also been amended to correct a typographical error and an error in grammar.



Conclusion

The applicant's invention as defined in the independent claims therefore includes a feature that is not found or suggested in the cited prior art, and that provides a substantial technical advantage. As such the applicant respectfully submits that the independent claims should be allowed, as well as the remaining, dependent claims that simply further narrow the definition of the invention.

Change of Attorney Name

Please note the enclosed letter concerning the change of the attorney's name from "Slusher" to "Pearce."

Date: 20 October 2003

Respectfully submitted,

Jeffrey Pearce

34825 Sultan-Startup Rd.
Sultan, WA 98294
Phone & fax: (360) 793-6687

Jeffrey Pearce
Reg. No. 34,729
Attorney for the Applicant